Meta-model Pruning

Sagar Sen, Naouel Moha, Benoit Baudry, Jean-Marc Jézéquel

Meta-model Pruning*

Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel

INRIA Rennes-Bretagne Atlantique, Campus universitaire de beaulieu,
35042 Rennes Cedex, France
{ssen,moha,bbaudry,jezequel}@irisa.fr

Abstract. Large and complex meta-models such as those of UML and its profiles are growing due to modelling and inter-operability needs of numerous stakeholders. The complexity of such meta-models has led to coining of the term *meta-muddle*. Individual users often exercise only a small view of a meta-muddle for tasks ranging from model creation to construction of model transformations. What is the *effective meta-model* that represents this view? We present a flexible *meta-model pruning* algorithm and *tool* to extract effective meta-models from a meta-muddle. We use the notion of *model typing* for meta-models to verify that the algorithm generates a *super-type* of the large meta-model representing the meta-muddle. This implies that all programs written using the effective meta-model will work for the meta-muddle hence preserving backward compatibility. All instances of the effective meta-model are also instances of the meta-muddle. We illustrate how pruning the original UML meta-model produces different effective meta-models.

Keywords: Meta-model pruning, GPML, DSML, UML, Kermeta, effective modelling domain, test input domain.

1 Introduction

Development of complex software systems using *modelling languages* to specify *models* at high-levels of abstraction is the philosophy underlying Model-Driven Engineering (MDE). There are two schools of thought that advocate the development of such modelling languages : *general-purpose modelling* and *domain-specific modelling*. General-purpose modelling is leveraged by modelling languages such as the Unified Modelling Language (UML)[1] with a large number of classes and properties to model various aspects of a software system using the same language. The UML superstructure consists of subsets of visual modelling languages such as UML use case diagrams, activity diagrams, state machines, and class diagrams to specify models of software systems. UML is also extensible using the *profiles mechanism* [2] to provide modelling elements from specific domains such as services, aerospace systems, software radio, and data distribution [3]. One

* The research leading to these results has received funding from the European Communitys Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 32–46, 2009.

of the primary advantages of the UML standard and its profiles is *inter-operability* between related domains in software development. On the other hand, domain-specific modelling promotes the construction of pure domain-specific modelling languages (DSMLs) [4]. One of the main disadvantages of a DSML is finding the ideal scope for its long term use. Identifying the scope involves abstracting DSML concepts in very early stages of its development. This leaves little room for adding concepts later in the lifetime of DSML. Despite the existence of several DSMLs general-purpose modelling languages (GPMLs) such as UML and its profiles are widely used to model complex software systems.

A major disadvantage of GPMLs such as the UML is its ever growing complexity and size. The widely accepted modelling language UML 2.0 has a specification document of about 1000 pages. The UML 2.0 *meta-model* used to specify the language contains 246 classes and 583 properties. The large number of classes and properties with several complex dependencies between them has led to the coining of the censorious term *meta-muddle* [5] to characterize huge GPMLs such as the UML. This criticism of UML can be attributed to the fact that it is an over-specification of the real *modelling domain* for a given application. For instance, if we intend to generate code from UML state machines there is no need to expose modelling elements for activity diagrams, or use case diagrams to the code generator. In practice, each application of the UML utilizes a subset of classes and properties in the UML. What is the *effective meta-model* that contains these required classes and properties and all its mandatory dependencies? This is the question that intrigues us and for which we provide a solution.

In this paper, we present a *meta-model pruning algorithm* that takes as input a large meta-model and a set of required classes and properties, to generate a target *effective meta-model*. The effective meta-model contains the required set of classes and properties. The term *pruning* refers to removal of unnecessary classes and properties. From a graph-theoretic point of view, given a large input graph (large input meta-model) the algorithm removes or prunes unnecessary nodes (classes and properties) to produce a smaller graph (effective meta-model). The algorithm determines if a class or property is unnecessary based on a set of rules and options. One such rule is removal of properties with lower bound multiplicity 0 and who's type is not a required type. We demonstrate using the notion of model typing that the generated effective meta-model, a subset of the large meta-model from a set-theoretic point of view, is a *super-type*, from a type-theoretic point of view, of the large input meta-model. This means that all programs written using the effective meta-model can also be executed for models of the original large meta-model. The pruning process preserves the meta-class names and meta-property names from the large input meta-model in the effective meta-model. This also implies that all instances (models) of the effective meta-model are also instances of the initial large input meta-model. All models of the effective meta-model are exchangeable across tools that use the large input meta-model as a standard. The extracted effective meta-model is very much like a transient DSML with necessary concepts for a problem domain at a given time. For example, we present an application of our algorithm to generate an

effective meta-model to specify test models for a model transformation. The model transformation is developed by the French Spatial Agency (CNES) to transform UML models to code for embedded systems.

The paper is organized as follows. In Section 2 we present the motivation for our work. We present related work in Section 3 that attempt to solve problems discussed in motivation. In Section 4 we present the meta-model pruning algorithm. We introduce model typing in Section 5 to show that the effective meta-model is indeed a super-type of the large meta-model. In Section 6 we present the application of meta-model pruning to obtain an effective meta-model for develop test models for a model transformation. We conclude and present future work in Section 7.

2 Motivation

The motivation for us to develop a meta-model pruning algorithm comes from observations made by us and others in various phases of the MDE process. We categorize our observations in the form of scenarios:

Scenario 1: Input Domain of Model Transformations. A large meta-model such as that of UML is the de facto input meta-model for a large number of model transformations or model processing programs/tools. However, many of these model transformations manipulate only a subset of the concepts defined in the input meta-model. There is a sparse usage of concepts in the input meta-model. For instance, code generators from UML state machines [6] normally use only the UML class diagram and UML state machine modelling elements. Therefore, often the *large meta-model is not the real input meta-model* of a model transformation. We illustrate this scenario in Figure 1 (a) where meta-model MM_{large} specifies a large set of models but a model transformation MT is developed to process only a subset of this large set.

Scenario 2: Chain of Model Transformations. A consequence of not defining the real input domain of a model transformation is the non-compatibility/mismatch of outputs and inputs between transformations in chain. Consider a sequence of model transformations as shown in Figure 1 (b). The output meta-model MM_o^a of model transformation MT_a is also the input meta-model MM_i^b for the next model transformation MT_b. However, we do not know if all models generated by MT_a can be processed by the model transformation MT_b as the concepts manipulated by the model transformations may be different. In [7], we identify this issue as one of the barriers to validate model transformations. Not identifying and dealing with this mismatch between the real input meta-model and real output meta-model can lead to serious software faults.

Scenario 3: Testing Model Transformations. Creating a model that conforms to a large meta-model does not always require all the concepts in the meta-model. For instance, if you want to create a model to test a model transformation of the large meta-model you may need to use only a small number of concepts. The entire large meta-model does not serve the purpose of creating test models for a certain sub-domain of the input meta-model. The large

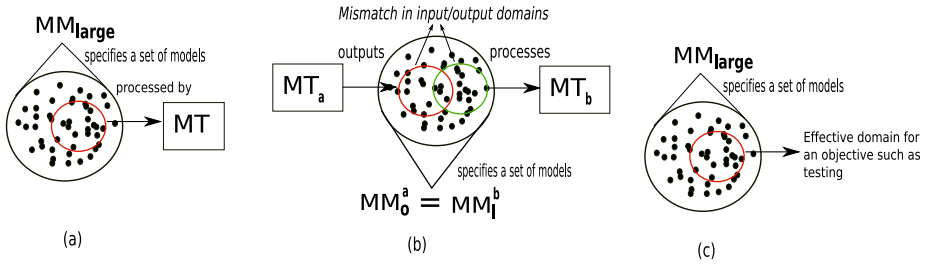

Fig. 1. Effective Meta-model Scenarios in Model Transformation Development

meta-model could pose a problem for a test model developer as she/he can be confused by the large number of concepts in the meta-model. In the context of automated testing, if you want to generate test models (such as using the tool Cartier [8] [9]) then you would want to transform the smallest possible input meta-model to a formal language for constraint satisfaction. Transforming the entire meta-model to a formal language will lead to a enormous constraint satisfaction problem. These large constraint satisfaction problems are often intractable. Solving smaller constraint satisfaction problems obtained from a small set of concepts and subsequently with fewer variables is relatively feasible.

Scenario 4: Software Process Modelling: Software process models contain several workflows. However, each workflow in a software process uses different sub-domains of a single shared meta-model such as the the UML. These workflows are often realized by different people and at different times. There are several software process methodologies that use the UML as the shared modelling language. The most popular of them is the Rational Unified Process (RUP) [10].

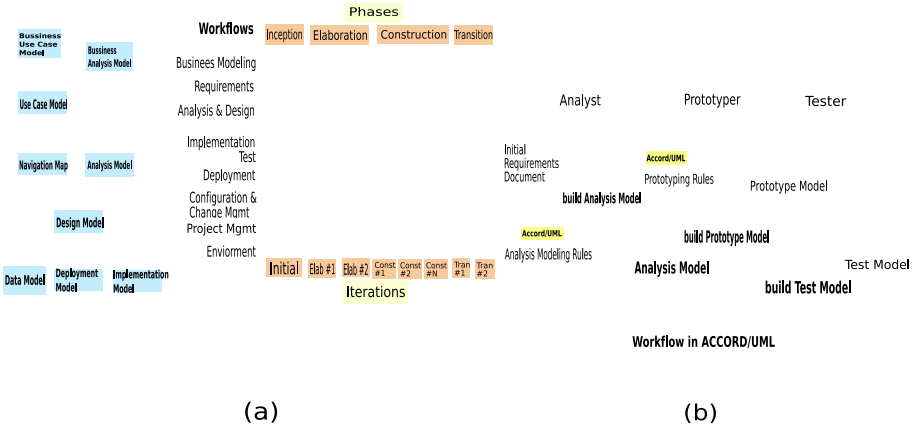

Fig. 2. (a) Workflows in RUP and its usage of UML (b) Workflow of ACCORD and its use of UML

Figure 2(a) shows the different workflows of RUP and the use of different subsets of UML for each workflow. Dedicated software processes such as ACCORD [11] use UML extended with domain-specific constructs to develop real-time systems. In Figure 2(b), we show the use of subsets UML in the ACCORD process. People involved in a certain workflow of a software process are exposed to concepts in the entire shared meta-model such as the UML instead of a subset of UML that represents their real work area. The access to unnecessary modelling elements to an engineer of a workflow could lead to errors in the software process.

The above scenarios are only some of the many possible scenarios where a large meta-model defines the modelling domain while only a sub-domain is in use.

3 Related Work

There has always been a need to define the effective modelling domain for a given objective in MDE. This is true especially in the case of using large GPMLs such as UML. In this section we present related work that deal with the problem of obtaining and using the effective modelling domain. We also pinpoint our contributions in this work.

Consider a fundamental task in MDE: Creating a model in a model editor such as in the Eclipse [12] environment. A popular editor for UML models is TOPCASED [13]. The tool can be used to create UML models such as class diagrams, state machines, activity diagrams, and use-case diagrams. If a modeller chooses to create class diagrams the tool presents the user with modelling elements for class diagrams such as classes and associations but not UML state machine modelling elements such as states and transitions. Therefore, the tool inherently prevents the modeller from using an unnecessary part of the UML meta-model. The *hard-coded* user interface in TOPCASED in fact presents the modeller with an effective modelling domain.

Model transformations on GPMLs such as UML are built for specific tasks and can process only a sub-domain of its huge input domain. To filter the input to a model transformation *pre-conditions* [14] are specified in a constraint language such as Object Constraint Language (OCL) [15] [16]. Graph transformation based model transformation languages specify pre-conditions to apply a graph rewriting rule on a left-hand side model pattern [17]. Both pre-condition contracts and patterns are specified on the entire input meta-model while they refer to only a sub-domain.

In the paper [5] Solberg et al. present the issue of navigating the meta-muddle notably the UML meta-model. They propose the development of Query/Extraction tools that allow developers to query the metamodel and to extract specified views from the metamodel. These tools should be capable of extracting simple derived relationships between concepts and more complex views that consist of derived relationships among many concepts. They mention the need to extract such views for different applications such as to define the domain of a model transformation and extracting a smaller metamodel from the concepts used in a model. Meta-modelling tools such as those developed by Xactium [18] and Adaptive Software [19] possess some of these abilities. The authors of [5] propose the

use of *aspects* to extract such views. However, the authors do not elaborate on the objectives behind generating such views.

In this paper, we present the following contributions emerging from our observations in MDE and survey of previous work:

- **Contribution 1:** We present a meta-model pruning algorithm to extract an effective meta-model from a large meta-model.
- **Contribution 2:** We present an application of model typing to *verify* that an effective meta-model is indeed a super-type of the large input meta-model. All programs written using the effective meta-model are valid also for the original large meta-model. Our approach preserves meta-concept names in the effective meta-model from the large meta-model and hence all instances of the effective meta-model are instances of the large input meta-model.

4 Meta-model Pruning Algorithm

This section presents the *meta-model pruning algorithm* to transform a input meta-model to a pruned target meta-model. We acknowledge the fact there can be an entire family of pruning algorithms that can be used to prune a large meta-model to give various effective meta-models. In this paper, we present a *conservative* meta-model pruning algorithm to generate effective meta-models. Our initial motivation to develop the algorithm was to help scale a formal method for test model generation [8] in the case of large input meta-models. Therefore, given a set of required classes and properties the rationale for designing the algorithm was to remove a maximum number of classes and properties facilitating us to scale a formal method to solve constraints from a relatively small input meta-model. The set of required classes and properties are inputs that can come from either static analysis of a transformation, an example model, an objective function, or can be manually specified. Given these initial inputs we automatically identify mandatory dependent classes and properties in the meta-model and remove the rest. For instance, we remove all properties which have a multiplicity 0..* and with a type not in the set of required class types. However, we also add some flexibility to the pruning algorithm. We provide options such as those that preserve properties (and their class type) in a required class even if they have a multiplicity 0..*. In our opinion, no matter how you choose to design a pruning algorithm the final output effective meta-model should be a supertype of the large input meta-model. The pruning algorithm must also preserve identical meta-concept names such that all instances of the effective meta-model are instances of the large input meta-model. These final requirements ensure backward compatibility of the effective meta-model with respect to the large input meta-model.

4.1 Algorithm Overview

In Figure 3, we present an overview of the meta-model pruning algorithm. The inputs to the algorithm are: (1) A source meta-model $MM_s = MM_{large}$ which is

also a large meta-model such as the meta-model for UML with about 246 Classes and 583 properties (in Ecore format) (2) A set of required classes C_{req} (3) A set of required properties P_{req}, and (4) A boolean array consisting of parameters to make the algorithm flexible for different pruning options.

The set of required classes C_{req} and properties P_{req} can be obtained from various sources as shown in Figure 3: (a) A static analysis of a model transformation can reveal which classes and properties are used by a transformation (b) The sets can be directly specified by the user (c) A test objective such as a set of partitions of the meta-model [20] is a specified on different properties which can be source for the set P_{req}. (d) A model itself uses objects of different classes. These classes and their properties can be the sources for C_{req} and P_{req}.

The output of the algorithm is a pruned effective meta-model $MM_t = MM_{effective}$ that contains all classes in C_{req}, all properties in P_{req} and their associated dependencies. Some of the dependencies are mandatory such as all super classes of a class and some are optional such as properties with multiplicity 0..* and whose class type is not in C_{req}. A set of parameters allow us to control the inclusion of these optional properties or classes in order to give various effective meta-models for different applications. The output meta-model $MM_{effective}$ is a subset and a super-type of MM_s.

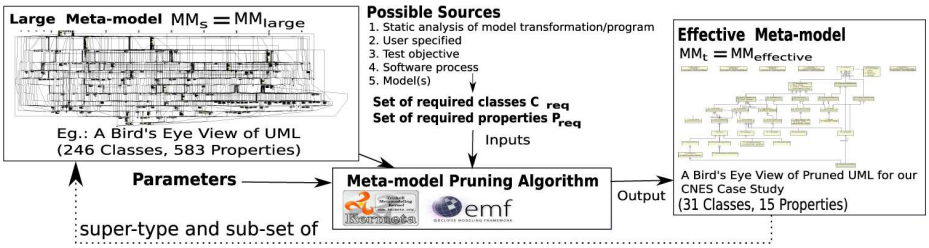

Fig. 3. The Meta-model Pruning Algorithm Overview

4.2 General Definitions

We present some general definitions we use for describing the meta-model pruning algorithm:

Definition 1. A primitive type b is an element in the set of primitives: $b \in \{String, Integer, Boolean\}$.

Definition 2. An enumeration type e is a 2-tuple $e := (name, L)$, where $name$ is a $String$ identifier, L is a finite set of enumerated literals.

Definition 3. A class type c is a 4-tuple $c := (name, P_c, Super, isAbstract)$, where $name$ is a $String$ identifier, P_c is a finite set of properties of class c, class c inherits properties of classes in the finite of classes $Super$ and $isAbstract$ is a $Boolean$ that determines if c is abstract.

Definition 4. A meta-model MM is a 2-tuple $MM := (T, P, Inv)$, where T is a finite set of class, primitive, and enumeration types, P is a set of properties, Inv is a finite set of invariants.

Type Operations: The operations on types used in this paper are: (a) $t.isInstanceOf(X)$ that returns true if t is of type X or inherits from X. (b) $t.allSuperClasses()$, if $t.isInstanceOf(Class)$, returns the set of all its super classes $t.Super$ including the super classes of its super classes and so on (multi-level).

Definition 5. A property p is a 7-tuple $p := (name, oC, type, lower, upper, opposite, isComposite)$, where $name$ is a $String$ identifier, oC is a reference to the owning class type, $type$ is a reference to the property type, $lower$ is a positive integer for the lower bound of the multiplicity, $upper$ is the a positive integer for the upper bound of the multiplicity, $opposite$ is a reference to an opposite property if any, and $isComposite$ determines if the objects referenced by p are composite (No other properties can contain these objects).

Property Operations: The operations on properties in this paper is $p.isConstrained()$ which returns $true$ if constrained by any invariant i such that $p \in i.P_I$. This is checked for all invariants $i \in MM.Inv$.

Definition 6. An invariant I is a 3-tuple $c := (T_I, P_I, Expression)$, where T_I is the set of types used in the invariant I and P_I is the set of properties used in I. An $Expression$ is a function of T_I and P_I that has a boolean value. The $Expression$ is often specified in a constraint language such as OCL [15].

Note: Throughout the section, we use the *relational dot-operator* to identify an element of a tuple. For example, we want to refer to the set of all types in a meta-model we use the expression $MM.T$,or $MM.P$ to refer to the set of all properties. Also, we do not consider user-defined meta-model *operations* or its argument signatures in our approach.

4.3 The Algorithm

The meta-model pruning algorithm (shown in Algorithm 1 has four inputs: (a) A source meta-model MM_s (b) Initial set of required types T_{req} (c) Initial set of required properties P_{req} (d) The top-level container class type C_{top}. (e) $Parameter$ which is a Boolean array. Each element in the array corresponds to an option to add classes or properties to the required set of classes and properties. In this paper, we consider three such options giving us a $Parameter$ vector of size 3.

The output of the algorithm is the pruned target meta-model MM_t. We briefly go through the working of the algorithm. The target meta-model MM_t is initialized with the source meta-model MM_s. The algorithm is divided into three main phases: (1) Computing set of all required types T_{req} in the meta-model, (2) Set of all required properties P_{req} in the meta-model (3) Removing all types and properties not that are not in T_{req} and P_{req}.

The first phase of the algorithm involves the computation of the entire set of required types T_{req}. The initial set T_{req} is passed as a parameter to the algorithm. We add the top-level container class C_{top} of MM_s to the set of required types T_{req} as shown in Step 2. In Step 3, we add the types of all required properties P_{req} to the set of required types T_{req}. In Step 4, we add types of all mandatory properties to T_{req}. Types of all properties with *lower bound greater than zero* are added to the set of required types T_{req} (Step 4.1). Similarly, if a property is constrained by an invariant in $MM.Inv$ then its type is included in T_{req} as shown in Step 4.2. If a property has an opposite type then we include the type of the opposite property, the owning class of the opposite property, and the type of the property to T_{req} in Step 4.3. The algorithm provides three options to add types of properties with lower multiplicity zero and are of type Class, PrimitiveType, and Enumeration respectively. The inclusion of these types is depicted in Steps 4.4, 4.5, and 4.6. The truth values elements of the $Parameter$ array determine if these options are used. These options are only examples of making the algorithm flexible. The $Parameter$ array and the options can be extended with general and user-specific requirements for generating effective meta-models. After obtaining T_{req} we add all its super classes across all levels to the set T_{req} as shown in Step 5.

The second phase of the algorithm consists of computing the set of all required properties P_{req}. Inclusion of mandatory properties are depicted from Step 6.1 through Step 6.5. In Step 6.1, we add all properties whose type are in T_{req} to P_{req}. In Step 6.2 we add all properties whose owning class are in T_{req} to P_{req}. In Step 6.3, we add properties with lower multiplicity greater than zero to P_{req}. If a property is constrained by a constraint in $MM.Inv$ we add it to P_{req} as depicted in Step 6.4. We add the opposite property of a required property to P_{req}. Finally, based on the options specified in the $Parameter$ array, the algorithm adds properties to P_{req} with lower multiplicity zero and other characteristics.

In the third phase of the algorithm we remove types and properties from MM_t. In Step 7, we remove all properties that are not in P_{req} (Step 7.1) and all properties who's types are not in T_{req} (Step 7.2). In Step 8, we remove all types not in T_{req}. The result is an effective meta-model in MM_t. In Section 5, we present *model typing* for meta-models to show that MM_t is a super-type of MM_s. As a result, any program written with MM_t can be executed using models of MM_s.

4.4 Implementation

The meta-model pruning algorithm has been implemented in Kermeta [21]. Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility(MOF) standard [22]. The tool supports input meta-models in the Eclipse Modelling Framework's (EMF) [12] Ecore meta-modelling standard. The tool with usage instructions is available for download [23].

Algorithm 1. metamodelPruning(MM_s, T_{req}, P_{req}, C_{top}, $Parameter$)

1. Initialize target meta-model MM_t
$MM_t \leftarrow MM_s$
2. Add top-level class into the set of required types
$T_{req} \leftarrow T_{req} \cup C_{top}$
3. Add types of required properties to set of required types
$P_{req}.each\{p|T_{req} \leftarrow T_{req} \cup p.type\}$
4. Add types of obligatory properties
$MM_t.P.each\{p|$
4.1 $(p.lower > 0) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
4.2 $(p.isConstrained(MM_t.Inv)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
4.3 $(p.opposite! = \phi) \implies \{T_{req} \leftarrow T_{req} \cup p.type, T_{req} \leftarrow T_{req} \cup p.opposite.type, T_{req} \leftarrow T_{req} \cup p.opposite.oC\}$
Option 1: Property of type Class with lower bound 0
if $Parameter[0] == True$ **then**
4.4 $(p.lower == 0\, and\, p.type.isInstanceOf(Class)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
end if
Option 2: Property of type PrimitiveType with lower bound 0
if $Parameter[1] == True$ **then**
4.5 $(p.lower == 0\ and\ p.type.isInstanceOf(PrimitiveType)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
end if
Option 3: Property of type Enumeration with lower bound 0
if $Parameter[2] == True$ **then**
4.6 $(p.lower == 0 and p.type.isInstanceOf(Enumeration)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}\}$
end if
5. Add all multi-level super classes of all classes in T_{req}
$MM_t.T.each\{t \mid t.isInstanceOf(Class) \implies t.allSuperClasses.each\ \{s|T_{req} \leftarrow T_{req} \cup s\}\}$
6. Add all required properties to P_{req}
$MM_t.P.each\{p|$
6.1 $(p.type \in T_{req}) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.2 $(p.oC \in T_{req}) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.3 $(p.lower > 0) \implies P_{req} \leftarrow P_{req} \cup p\}$
6.4 $(p.isConstrained(MM_t.Inv)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.5 $(p.opposite! = \phi) \implies \{P_{req} \leftarrow P_{req} \cup p, P_{req} \leftarrow P_{req} \cup p.opposite\}$
Option 1: Property of type Class with lower bound 0
if $Parameter[0] == True$ **then**
6.6 $(p.lower == 0\, and\, p.type.isInstanceOf(Class)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
end if
Option 2: Property of type PrimitiveType with lower bound 0
if $Parameter[1] == True$ **then**
6.7 $(p.lower == 0\ and\ p.type.isInstanceOf(PrimitiveType)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
end if
Option 3: Property of type Enumeration with lower bound 0
if $Parameter[2] == True$ **then**
6.8 $(p.lower == 0 and p.type.isInstanceOf(Enumeration)) \implies \{P_{req} \leftarrow P_{req} \cup p\}\}$
end if
7. Remove Properties
$MM_t.P.each\{p|$
7.1 $p \notin P_{req} \implies (t.P \leftarrow t.P - p)$
7.2 $p.type \notin T_{req} \implies (t.P \leftarrow t.P - p)\}$
}
8. Remove Types
$MM_t.T.each\{t|t \notin T_{req} \implies MM_t.T \leftarrow MM_t.T - t\}$

5 Model Typing

In the section we describe the notion of *model typing*. We use model typing to verify that meta-model pruning algorithm indeed generates a super-type of the input meta-model. Model typing corresponds to a simple extension to object-oriented typing in a model-oriented context [24]. A model typing is a strategy for typing models as collections of interconnected objects while preserving type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type groups and type group matching [25]. The matching relation, denoted <#, between two meta-models defines a function of the set of classes they contain according to the following definition:

> Metamodel *M'* matches another metamodel *M* (denoted *M' <# M*) iff for each class *C* in *M*, there is one and only one corresponding class *C'* in *M'* such that every property *p* and operation *op* in *M.C* matches in *M'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *M.C*.

This definition is adapted from [24] and improved here by relaxing the constraint related of the name-dependent conformance on properties and operations.

Let's illustrate model typing with two metamodels *M* and *M'* given in Figures 4 and 5. These two metamodels have properties and references that have different names. The metamodel *M'* has additional elements compared to the metamodel *M*.

> *C1 <# COne* because for each property *COne.p* of type *D* (namely, *COne.name* and *COne.aCTwo*), there is a matching property *C1.q* of type *D'* (namely, *C1.id* and *C1.aC2*), such that *D' <# D*.
>
> Thus, *C1 <# COne* requires *D' <# D*:
>
> - *COne.name* and *C1.id* are both of type *String*.
> - *COne.aCTwo* is of type *CTwo* and *C1.aC2* is of type *C2*, so *C1 <# COne* requires *C2 <# CTwo*. And, *C2 <# CTwo* is true because *CTwo.element* and *C2.elem* are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [26]. The interested reader can find the details of matching rules used for model types in [26].

In Section 6, we illustrate the use of model typing integrated in the model transformation language Kermeta. We show that transformations written using the effective meta-model are also valid for models of the original large meta-model.

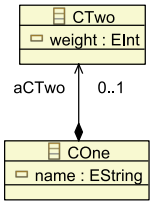

Fig. 4. Metamodel *M*

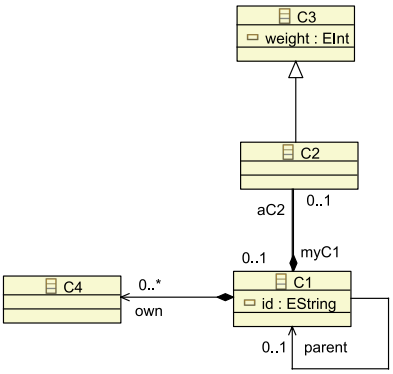

Fig. 5. Metamodel *M'*

6 Application

We apply the meta-model pruning algorithm to generate an effective meta-model to specify test models for model transformations. The model transformation in our case study is from the French National Space Agency (CNES) to generate embedded systems code from a set of input models. The project is sponsored by the DOMINO project of ANR. We do not discuss the transformation in detail in this paper. We, however, highlight that the transformation uses a subset of UML Activity diagram models. Our algorithm extracts an effective meta-model with the ultimate objective of testing the transformation. Testing can be done either by manually specifying test models or automatically generating them based on the technique in [8]. We do not elaborate on the testing phase in this paper.

```
package cnesTransfoMain;
require "http://www.eclipse.org/uml2/2.1.2/UML"
class Main {
    operation main() : Void is do
        var rep : EMFRepository init EMFRepository.new
        var res : kermeta::persistence::EMFResource
        res ?= rep.getResource("model.uml")
        var inputModel : uml::Model  //Input UML Model
        model ?= res.one
        var transfo : cnesPackage::Transfo<uml::UmlMM>
                      init cnesPackage::Transfo<uml::UmlMM>.new
        transfo.generateCode(inputModel)
     end }
-------------------------
package cnesPackage;
require UMLCNES;
class Transfo<MT : UMLCNES> { // Code generator...
    operation generateCode( source : MT::Model)  : Void is do
    ... end }
```

Listing 1. Kermeta Transformation to Demonstrate use of Effective Meta-model

The result of executing the algorithm with *no options* (no parameter specified) is the bare-minimum effective meta-model shown in Figure 6. A bare minimum effective meta-model , in our case, is sufficient to specify input models for the transformation. The meta-model is generated using an initial set of required classes C_{req}. All elements of C_{req} are provided as input the the pruning algorithm in the set T_{req} such that $C_{req} \in T_{req}$. The classes in C_{req} are shown within red

boxes in Figure 6. The top-level class $C_{top} = Model$ is specified in a green-dashed box. In the pruned meta-model we observe that all disjoint subgraphs of the UML meta-model are removed such as UML State Machines, UML Class Diagrams, and UML Use Case Diagrams preserving only a subset of UML Activity diagram.

We call the resulting $MM_{effective}$ of UML, UMLCNES. We can verify that UMLCNES is a *super-type* of UML using the notion of model types described in Section 5. The type checking rules for model types has been integrated into the typing system of the modelling and model transformation language Kermeta [21]. We can write a transformation using UMLCNES as the input domain as shown in listing 1. The package *cnesTransfoMain* calls the *generateCode* operation (in package *cnesPackage*) with an UML input model. However, the transformation is defined for the UMLCNES meta-model. The transformation will still execute since UMLCNES is a super-type of UML. Test models can also be developed as instances of UMLCNES and transformed to UML without loss of information.

The pruning algorithm is *flexible*. We briefly illustrate this by pruning UML for the different options presented in the paper. In Table 1 we summarize the number of classes and properties for the different options of the meta-model pruning algorithm. The algorithm can be used to generate different effective meta-models with various applications. For example, another option that is not dealt with in this paper could be inclusion of all possible containers of a property to the set of required types. Options can be used to relax or tighten the pruning for applications where model transformations may evolve and use more concepts that initially perceived.

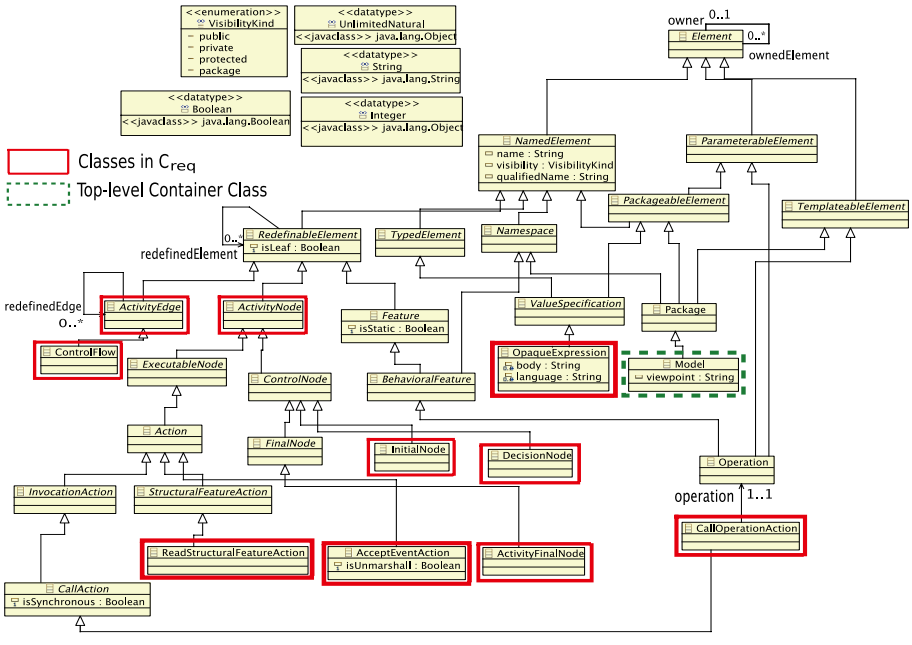

Fig. 6. The Effective UML Activity Diagram Meta-model for the CNES Case Study

Table 1. Meta-model Pruning Results for Options

	Original Uml	No Option	Option 1	Option 2	Option 3
Number of Classes	246	31	31	31	31
Number of Properties	583	15	26	30	30

7 Conclusion

Deriving effective modelling domains is an ubiquitous need in MDE. There are several existing ways such as invariants, pre-conditions and hard-coded knowledge in model editors such as TOPCASED to obtain some form of an effective modelling domain. Most of these approaches patch up the modelling domain with constraints or code to obtain a constrained or effective modelling domain. In this paper, we present an algorithm to extract an effective meta-model from a large meta-model via pruning the large meta-model. Very much like extracting the meta-model of a transient DSML. The input to the algorithm includes the large meta-model and a set of required classes and properties. The algorithm finds all mandatory dependencies between these required concepts. It then prunes the large meta-model such that only the required concepts and its mandatory dependencies are preserved. The flexible algorithm also allows inclusions of non-mandatory properties. The effective meta-model typically has fewer classes and properties compared to the input meta-model and is a super-type of the input meta-model. Therefore, any program written for the effective meta-model will also accept models of the large meta-model. In the future, we would like to integrate the meta-model pruning algorithm to dynamically generate an effective meta-model in MDE tool chains such as editors and transformations. There is also scope for adding more options to control the generation of an effective meta-model for various objectives.

References

1. OMG: UML 2.0 Specfication, `http://www.omg.org/spec/UML/2.0/`
2. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to uml profiles. UPGRADE, European Journal for the Informatics Professional 5(2), 5–13 (2004)
3. OMG: UML Profile Catalog, `http://www.omg.org/technology/documents/profile_catalog.htm`
4. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35(6), 26–36 (2000)
5. Solberg, A., France, R., Reddy, R.: Navigating the metamuddle. In: Proceedings of the 4th Workshop in Software Model Engineering, Montego Bay, Jamaica (2005)
6. Niaz, I.A., Tanaka, J.: Code generation from uml statecharts. In: Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey, pp. 315–321 (2003)
7. Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM (2009)
8. Sen, S., Baudry, B., Mottu, J.M.: On combining mullti-formalism knowledge to select test models for model transformaion testing. In: ACM/IEEE International Conference on Software Testing, Lillehammer, Norway (April 2008)

9. Sen, S., Baudry, B., Mottu, J.M.: Automatic model generation strategies for model transformation testing. In: ICMT (2009)
10. Kruchten, P.: The Rational Unified Process: An Introduction, 3rd edn. Addison-Wesley Professional, Reading
11. Phan, T.H., Gerard, S., Terrier, F.: Real-time system modeling with accord/uml methodology: illustration through an automotive case study. In: Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL 2003, pp. 51–70 (2004)
12. Frank, B.: Eclipse Modeling Framework. The Eclipse Series, vol. 1. Addison-Wesley, Reading (2004)
13. Farail, P., Gaufillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In: Embedded Real Time Software (ERTS), Toulouse, February-May (2006)
14. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. 20(5), 42–45 (2003)
15. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07 (2007)
16. Lagarde, F., Terrier, F., André, C., Gérard, S.: Extending ocl to ensure model transformations, pp. 126–136 (2007)
17. Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (October 2005)
18. Limited, X.: Language driven development and xmf-mosaic. Whitepaper (2005)
19. Inc., A.: `http://www.adaptive.com/`
20. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Towards dependable model transformations: Qualifying input test data. Journal of Software and Systems Modeling, SoSyM (2007)
21. Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
22. OMG: Mof 2.0 core specification. Technical Report formal/06-01-01, OMG (April 2006) OMG Available Specification
23. Sen, S.: Meta-model pruning kermeta implementation, `https://www.irisa.fr/triskell/softwares-fr/protos/metamodelpruner/`
24. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) 6(4), 401–414 (2007)
25. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electronic Notes in Theoretical Computer Science 20, 50–75 (1999)
26. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)